Performance of Node.js Backend Application Frameworks. An Empirical Evaluation

Paweł Kuffel
Poznań University of Technology
Poznań, Poland *pawel@kuffel.io*

Bartosz Walter
Poznań University of Technology
Poznań, Poland *bartosz.walter@cs.put.poznan.pl*

Abstract

The Node.js ecosystem features a large number of backend application frameworks with diverse performance-related characteristics. Since performance is a critical factor for modern-world applications, they need to be carefully examined. This paper analyses performance of various frameworks and investigates changes in performance characteristics under sustained load, grounded in the analysis of runtime environment internals. Results indicate significant differences in application throughput obtained by various Node.js frameworks with a particular characteristic shared between all of them, yet absent in baseline SUTs implemented in other programming languages.

Keywords: Node.js, performance, benchmark, application framework, JavaScript

1. Introduction

Node.js is a cross-platform, MIT-licensed JavaScript runtime environment that enables JavaScript execution outside of a web browser. It is commonly used for server-side scripting and running backend applications. Since its 2009 release, Node.js' popularity has soared: the Stack Overflow Annual Developer Survey identified Node.js as the most popular technology in its category since 2017.

Node.js's popularity resulted in the emergence of a vast ecosystem of third party modules available via its package manager, `npm`. As of March 2024, it hosted over 2.7 million modules, surpassing the combined offerings of Maven Central (Java), Nuget (.NET), Packagist (PHP), and Python Package Index. On the wave of Node.js' popularity, several derivative backend frameworks have been developed. They exploit various features offered by the parent runtime, offering diverse performance and various functionalities, including request routing, user authentication, or error handling.

While differences between Node.js and other technologies were thoroughly examined, little is known about differences between various Node.js-based frameworks. Therefore, there is a need to conduct a multi-dimensional comparison of such frameworks, to identify their strengths and weaknesses in various scenarios. In this paper we compare performance of applications built using various Node.js-based frameworks, and investigate changes in their performance characteristics over time under sustained load. We also discuss how the underlying implementation and optimization techniques of Node.js could explain the observed irregularities in results.

The remainder of the paper is structured as follows. In Sec. 2 we discuss the architectural and functional details of the Node.js runtime environment. Then, Sec. 3 presents literature

This work has been supported by Poznań University of Technology under the 0311/SBAD/0750 project.

review. Sec. 4 outlines the experimental design, and Sec. 5 presents and discusses the outcomes, followed by analysis of the threats to validity in Sec. 6, and conclusions in Sec. 7.

2. Background

Certain aspect of Node.js internal implementation may be relevant to the performance characteristics of applications executed in that runtime environment. This section aims to describe selected details of the Node.js's architecture, focusing on the event loop concurrency model, hidden classes, inline caching and selective machine code generation.

2.1. Event Loop Concurrency Model

Traditional web server implementations use a *thread-per-connection model*, in which the main thread listens for incoming connections, and each connection is off-loaded to a separate, sequential thread for processing. While this approach simplifies development, such model may not scale well under load, as each active connection requires a separate thread to be managed by the operating system.

The event loop concurrency model addresses the shortcomings of the thread-per-connection model. Instead of spawning a separate thread per connection, this model awaits for changes across all connections and processes them in a single thread. This requires a different set of OS-level primitives for I/O multiplexing, such as `epoll` in Linux, `kqueue` in MacOS or I/O Completion ports in Windows.

Using `epoll`, a server can replace all connection threads with a single thread. Any operation that would otherwise be a blocking call (such as a database query) is executed asynchronously and once it is completed, its results are processed sequentially in an event loop iteration. In Node.js, the event loop implementation is provided by Libuv, a C library that abstracts platform-specific asynchronous I/O operations, reducing OS-specific branching within Node.js source code [8].

2.2. The V8 JavaScript engine

The V8 JavaScript engine is embedded in Node.js as its largest dependency. It consists of numerous modules working towards parsing, executing, and optimizing the JavaScript that is provided to it as an input by the process in which the V8 is embedded. The incoming source code is first parsed in order to produce an abstract syntax tree, which is later translated into bytecode by the Ignition module. During bytecode execution, various metrics are being collected, based on which the V8 might decide to optimize a particular unit of code using the Turbofan module.

2.3. Hidden classes

In statically typed languages objects have a predefined structure, allowing property access through fixed memory offsets, leading to efficient operations. In contrast, JavaScript, being dynamically typed, allows properties to be dynamically added or removed, necessitating the use of hashmaps for object implementation. This approach, however, incurs performance and memory overheads due to dynamic property lookups.

To mitigate these inefficiencies, the V8 JavaScript engine utilizes a concept called *hidden classes*. Despite the name, hidden classes are not related to traditional object-oriented programming classes; instead, they provide a structure to JavaScript objects that mimics the fixed layout of objects in statically typed languages. Hidden classes store the memory offsets of properties, enabling quicker access similar to that in statically typed environments.

2.4. Inline caching

Usage of hidden classes in the V8 JavaScript engine enables implementation of an optimization technique called *inline caching*. It is based on an observation that subsequent invocations a given function tend to have repeating object shapes in the call arguments. If such pattern is detected, the compiler/interpreter can cache results of expensive operations, the value of which is known to be the same for every object with a given hidden class, i.e., determining the offset of a particular property.

Inline caching in V8 operates by tracking all instances in the executed Ignition bytecode in which an object's properties are accessed. Each *Object Access Site* is associated with an entry in the *IC Vector*, which records hidden classes observed at these sites for specific functions. When objects with the same hidden class are repeatedly passed to a function, the relevant IC slot becomes *monomorphic*, storing a cached property offset. This allows the engine to bypass full property lookups during subsequent operations, provided that hidden class of the input matches the one recorded in the IC slot.

2.5. Selective machine code generation with Turbofan

Turbofan, the optimizing compiler for the V8 JavaScript engine, selectively translates Ignition bytecode into optimized machine code. This process is informed by type feedback and runtime profiling, which helps to identify performance bottlenecks by monitoring code execution intervals.

JavaScript's versatility allows for the same syntax to support multiple operations, such as the plus (+) operator functioning for both arithmetic and string concatenation, resulting in performance costs due to runtime type checks. Turbofan uses type feedback to speculatively optimize functions, generating machine code that anticipates specific object types. If actual parameter types differ from these expectations, the optimized code is discarded, reverting to Ignition bytecode through *de-optimization*.

3. Related work

3.1. Performance of Node.js-related applications

Since its release Node.js has become a viable alternative to traditional web development technologies, offering competitive performance and scalability [13], [1], [2]. Over the years, due to newly added optimisations end extensions, it remained among the most popular solutions for serving dynamic content on the web.

On the other hand, little is known about the relative comparison of various Node.js-based frameworks. Their authors and developers sometimes report results of experiments carried out against systems under test (SUTs) implemented in competing frameworks, but such experiments usually involve only a single, simple load testing scenario with a static HTTP response returned by a single endpoint and only report the average number of requests per second, without measuring changes in performance characteristics over the duration of the load test.

3.2. Tools and metrics

Among testing tools, Apache JMeter remains the most commonly used utility, outperforming other popular utilities [20]. LoadRunner is another tool used for facilitating load testing experiments [3, 10, 23, 7, 17].

One of metrics used in performance studies is response time (also known as *request latency*). Usually mean or median values are considered [14], [18], [17], accompanied by tail values [9], [21], [6]. Some studies also mention the average number of completed *requests per second* (RPS) as a less complex metric [14], [12], [19]. Other popular metrics refer to CPU, memory

and bandwidth [7], [17].

3.3. Testing environment

Different settings are in use and none is preferred. Several works advocate for conducting end-to-end load tests in a public cloud environments [14], e.g., Microsoft Azure VMs [12], which are more representative for real-life scenarios. Another popular setup relies on two locally connected machines: one for load generation and another for running the SUT [5], [11], [17, 4], but variations are also in use, e.g., *peer-to-peer load testing* with multiple load generation machines targeting a single SUT machine [15], [9] or with VMs deployed on a single computer [22], [19]. Finally, both the load driver and SUT application are also deployed on the same machine [7], [23].

4. Approach

4.1. Research questions and hypotheses

With a large ecosystem of Node.js backend frameworks available for developers to choose from, we aim to compare the performance characteristics of applications built using such dependencies in various testing scenarios. Furthermore, given that optimization techniques discussed in Sec. 2 are being deployed incrementally over the executed JavaScript program lifespan, we conjecture that Node.js server-side applications improve their performance over time under sustained load. Specifically, we aim at answering following questions:
RQ1 What are the performance differences among applications developed using various Node.js backend frameworks in different scenarios?
RQ2 Do Node.js backend applications improve their performance characteristics over time under sustained load?

To provide the answers, we present an empirical study that involves a load test of applications developed using various Node.js-based frameworks, and running a number of typical execution scenarios. We also discuss how various optimizations offered by the frameworks contribute to the observed result.

4.2. Experimental setup

For comparing the performance of frameworks we used a *synthetic benchmarking* model. Instead of creating real-world applications with simulated workload that mimics actual users' behaviour, individual functions of the SUT are tested in isolation, which helps to identify the impact of each factor. Based on that, we designed five testing scenarios that have been used in the study:

- *Static JSON* – responding with a static JSON payload (measuring framework's performance in the area of request routing, response writing and JSON serialization).
- *Fibonacci* – responding with a recursively-computed value of the 23rd element of the Fibonacci sequence (simulating a CPU-intensive request).
- *Redis* – retrieving a single key from a local Redis instance over a TCP connection and sending it back to the client to simulate asynchronous I/O workload.
- *Error* – reading a parameter from the request path and performing a division by zero if the parameter value is 1 (resulting in HTTP response with code 500; testing framework-specific exception handling).
- *Validation* – checking if the request query string contains two keys (integer and a string with a minimum length of 5; testing framework's built-in request validation features).

Table 1. Frameworks selected for the study

Framework	#forks	#stars	Keywords	URL at GitHub
Flask (Python)	15.9K	66.2K	microframework, extensibility	`/pallets/flask`
Feathers	736	16.9K	SOA, AOP, opinionated	`/feathersjs/feathers`
NestJS (Express)	7.4K	64.1K	Spring, AngularJS	`/nestjs/nest`
Express	13.6K	63.7K	minimal, robust, flexible	`/expressjs/express`
AdonisJS	0.6K	15.4K	all-in-one, MVC, custom HTTP	`/adonisjs/core`
Hapi	1.3K	14.5K	security, simplicity, stability	`/hapijs/hapi`
Fastify	2.2K	30.5K	performance	`/fastify/fastify`
Gin (Go)	7.8K	75.3K	performance, extensibility	`/gin-gonic/gin`

4.3. Evaluated frameworks

Due to a large variety of Node.js frameworks, we employed three generic criteria to select subjects: (1) *popularity*, measured via the number of GitHub stars and `npm` package downloads, (2) *versatility* – general-purpose frameworks were preferred over domain-specific ones, (3) *availability of documentation* and other resources useful in development of SUT applications. The frameworks are presented in Table 1.

For each framework we developed a dedicated SUT application. Furthermore, to put the collected performance metrics into perspective, we developed also two additional SUT applications exposing the same HTTP API: one in the Go programming language (using Gin framework) and another one in Python (Flask framework), which will serve for a reference.

4.4. Tooling

To prevent a potential bottlenecking effect, a preliminary experiment was performed comparing JMeter's metrics with those obtained via WRK [1] – another popular, low-overhead HTTP benchmarking utility. Across 5 testing scenarios, WRK has outperformed JMeter in terms of the number of generated requests per second, with the largest gap of 134% observed in the static JSON scenario (Go/Gin SUT), which was a strong indication of bottlenecking.

However, due to WRK's inability to collect raw request logs (it only presents request latencies in a histogram), we decided to develop a purpose-built, custom load testing tool called Gocannon[2]. It was written in Go, as the use of goroutines allowed for a relatively easy development of multi-threading features while relying on efficient, epoll-based I/O features delivered by the underlying runtime.

4.5. Testing procedure

The entire testing procedure was automated using a bash script. Each of the SUT applications (7 Node.js programs, 1 compiled Go binary and 1 Python project) was a subject to five testing scenarios. To reduce bias, each test was repeated 10 times, and we report the average value. To minimize network-related inconsistencies, all components (Gocannon, the SUT and a Redis database instance) were running in the same machine.

Additional measures were undertaken to ensure constant hardware performance over the entire duration of the load test. These included disabling Intel Turbo Boost, Enhanced Intel SpeedStep Technology and frequency scaling via Linux `cpupower` package (to ensure a constant CPU clock of 4 GHz). To confirm that the Redis instance is not becoming a bottleneck, a benchmark was conducted using the *redis-benchmark* utility, which showed that without instruction pipelining, the system was able to sustain 130k key lookups per second.

[1] `https://github.com/wg/wrk`
[2] `https://github.com/kffl/gocannon`

Table 2. Mean number of requests per second by framework and load testing scenario

Framework	Static JSON	Fibonacci	Redis	Validation	Error
Flask (Python)	2 159.78	148.79	1 954.79	2 047.19	1 655.90
Feathers	3 158.17	1 469.11	3 506.55	3 056.99	2 571.10
NestJS (Express)	5 303.64	1 788.35	5 843.45	4 342.29	2 384.03
Express	5 850.01	1 858.43	6 264.94	4 284.94	5 388.53
AdonisJS	19 395.50	2 533.78	17 051.89	9 820.35	6 292.51
Hapi	21 782.20	2 566.75	21 731.95	15 945.76	9 314.06
NestJS (Fastify)	22 597.08	2 432.39	22 348.36	15 721.51	4 110.02
Fastify	25 737.06	2 571.89	22 976.41	23 539.39	11 484.62
Gin (Go)	42 039.97	5 743.55	29 731.90	34 242.94	1 521.76

5. Results and discussion

5.1. Comparison of Node.js backend application frameworks performance

Results showing the number of requests per second handled are presented in Table 2. The reported time is a mean value of 10 runs, as designed in Sec. 4. Additionally, the results are presented in Fig. 1.

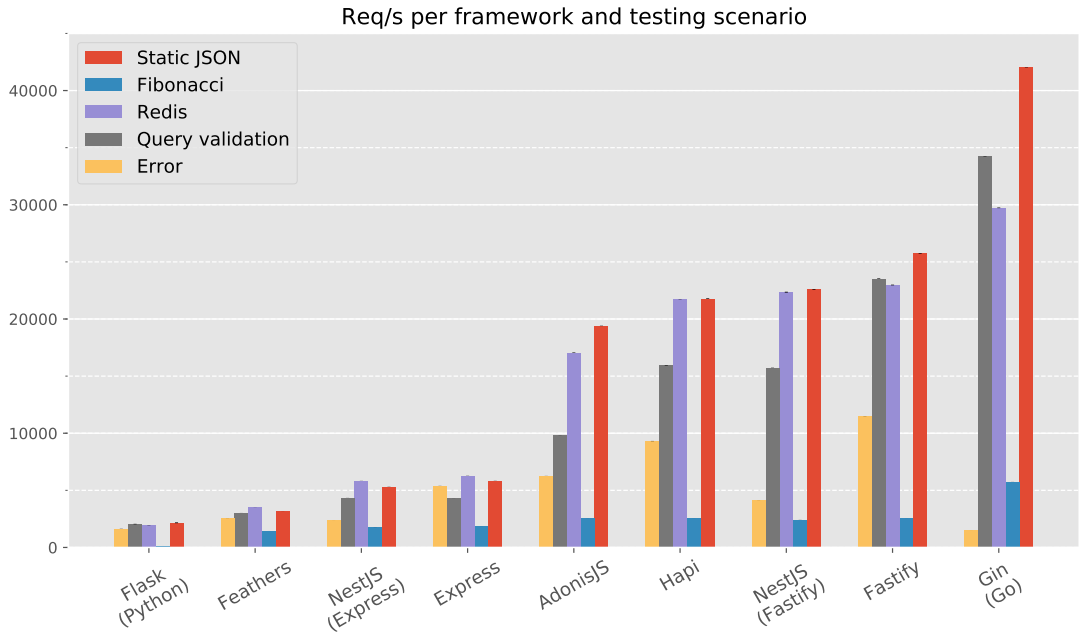

Fig. 1. Mean number of completed requests per second by framework and load testing scenario

Based on that, we provide an answer to RQ1. Details and possible explanations for the observed results are further discussed in Sec. 5.2.

The examined application frameworks could be grouped into two categories with respect to their performance. The first one includes Express, NestJS, and Feathers, which showed lower performance metrics compared to other SUTs. The added complexity of NestJS and Feathers, which are built atop of Express, further reduced their performance. The second group, with AdonisJS, Hapi, NestJS, and Fastify, displayed superior performance in all scenarios.

Despite JavaScript's dynamic typing, top-performing Node.js SUTs maintained robust performance, even against Go-based applications. Notably, Fastify was just 23% less efficient than Gin in a Redis scenario, managing 22,976 requests per second compared to Gin's 29,731. Conversely, Python's Flask lagged significantly, performing worst in four out of five tests.

5.2. Framework performance in different scenarios

In testing scenarios, the Go/Gin backend excelled in static JSON load tests, handling over 42,000 requests per second. However, in CPU-intensive Fibonacci load tests, performance differences among Node.js frameworks narrowed. All frameworks performed similarly, with the slowest (AdonisJS) achieving results within 5% of the fastest (Fastify). Python's Flask struggled in these scenarios, serving requests at a rate 10 times slower than the slowest Node.js framework using Feathers. In contrast, Go's Gin served 120% more requests than the fastest Node.js framework in Fibonacci tests, highlighting Go's efficiency as a compiled, statically-typed language.

The Redis test, which involved asynchronous I/O operations, showed that slower Node.js frameworks (Express, Feathers, and NestJS with Express) performed better than in static JSON tests, indicating a correlation between these scenarios.

In the querystring validation test, Fastify's schema validation (based on `ajv`) was the most efficient among Node.js backend frameworks. While Hapi and NestJS (using `joi` and `class-validator` respectively) produced comparable results, NestJS's validation outperformed AdonisJS significantly, with AdonisJS serving just over half the requests compared to its performance in the static JSON test. This suggests a substantial overhead in AdonisJS's validation process.

In the error handling load tests, Express.js showed minimal overhead, with an 8% drop in requests per second, compared to the static JSON test. However, NestJS exhibited significant performance drops in this scenario, regardless of whether it used Fastify or Express, pointing to the impact of its exception filters.

An interesting finding is that the SUT implemented in Go with the Gin framework served the fewest requests per second in the error test, despite outperforming in other four load testing scenarios. This could be linked to Go's unique error handling approach, which involves *returning* errors rather than *throwing* them, and the possibility of a panic that disrupts function execution. Gin uses `recover()` in its middleware to handle panics and log errors, potentially slowing performance. To explore this, a modified Go SUT with a simpler recovery middleware that omits error log tracing was tested, showing no significant overhead from the panic/recover mechanism itself. This test result suggests the initial performance dip was mainly due to the resource-heavy process of iterating over stack frames accessed via the `runtime` library in order to log stack traces in Gin's error handling.

5.3. Changes in application performance over time under sustained load

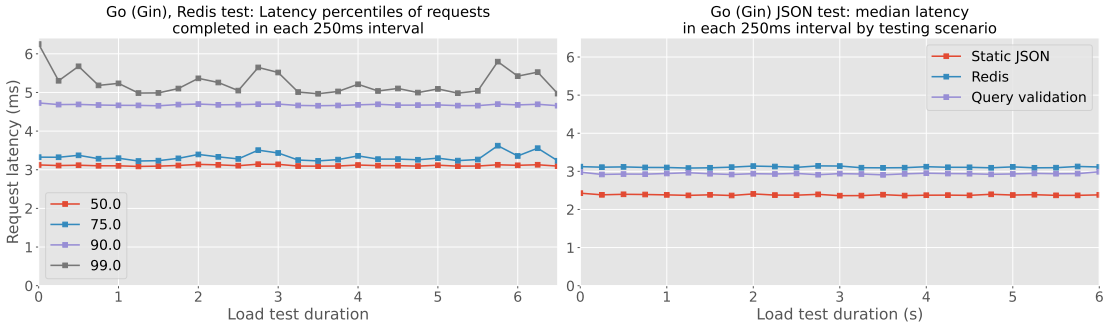

Fig. 2. Performance metrics of the Go (Gin) SUT in selected testing scenarios

To explore changes in application performance under sustained load, the request latency during load tests was analyzed. For the Go/Gin SUT, as shown in Fig. 2, the request latency remained consistent across various load testing scenarios, indicating stable throughput without significant fluctuations. The Python/Flast baseline SUT implementation exhibits similar (stable) behaviour under load to that of the Go backend application, albeit with a considerable performance gap.

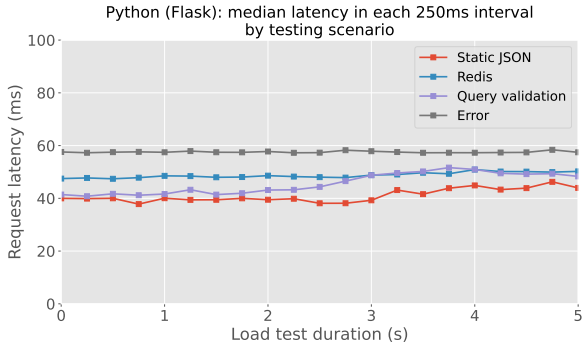

Fig. 3. Median response time over load test duration of the Python/Flask SUT in four load testing scenarios

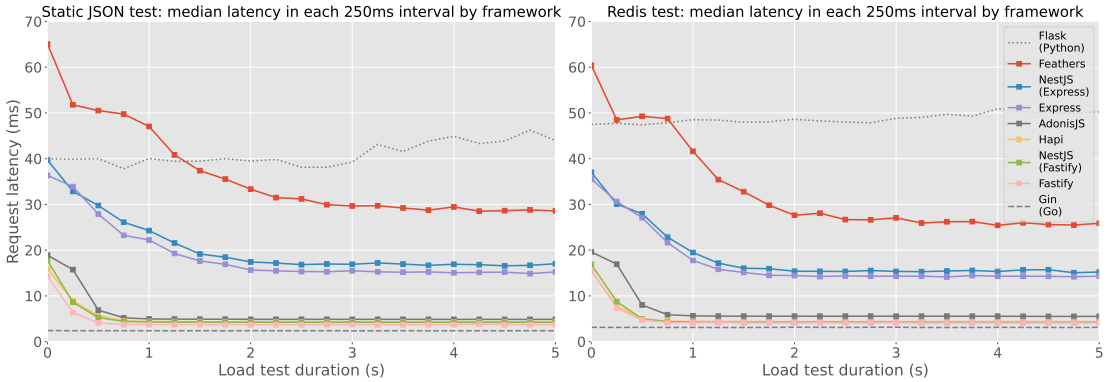

Fig. 4. Median request latency over the Static JSON/Redis load test duration by Node.js framework

Unlike SUTs implemented in Go and Python, Node.js frameworks displayed a decrease in request latency over time in load tests, particularly in the Redis scenario as presented in Fig. 4. High-performance Node.js frameworks (Fastify, Hapi, and NestJS with Fastify adapter) stabilized their latency after about 750 milliseconds, while frameworks based on Express.js required around 2.5 seconds to achieve stable latency, as seen in both Redis and static JSON load tests. A similar trend can be observed in the results of static JSON load testing scenario.

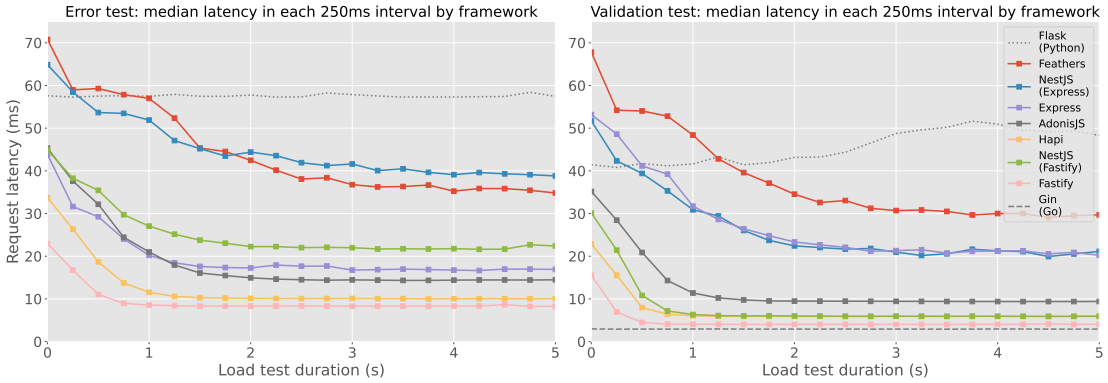

Fig. 5. Median request latency over the error/validation load test duration by Node.js framework

During the error handling tests, Fastify's performance was consistent with other scenarios, maintaining low latency, while NestJS, using its exception filters, exhibited a noticeable latency increase, especially with the Express.js dependency. This led to slower stabilization of median request latency compared to other Node.js frameworks like Hapi and Fastify, as shown in Fig. 5.

Express.js demonstrated a faster stabilization at around 1.5 seconds, quicker than in other scenarios. Concluding, the tendency for the median request latency to drop over the first few seconds of the test was observed in every scenario executed against every Node.js SUT.

The time it takes for median request latency to reach a stable value since the beginning of the test is correlated with the overall SUT performance in the load test. As depicted in Fig. 6, the lower the final median request latency obtained in latter time-frame of the load test, the sooner that the observed flattening occurs. Same is true for the 75th, 90th and 99th percentile values.

The performance of Node.js backend applications tends to improve over time under load, confirming our initial conjecture and providing an answer to RQ2. This behavior is particularly relevant in high-throughput environments where applications are scaled horizontally behind load balancers using algorithms like Round-Robin or Least-Connections [16]. Since newly added instances may initially perform less effectively, the Least-Connections method, which routes traffic to the least busy instance, is recommended to prevent overloading these new instances.

Regarding load testing, since Node.js applications show decreasing request latency until a stable point is reached, we advise conducting prolonged load tests. This approach helps account for the initial warm-up phase, providing a more accurate measure of the throughput.

The study highlights significant performance disparities among Node.js frameworks. Express.js, despite its popularity, shows poor performance relative to newer frameworks with less overhead. This impacts frameworks like NestJS and Feathers that rely on Express.js, resulting in lower performance metrics. Conversely, frameworks such as Fastify, Hapi, and AdonisJS exhibit robust load resistance, comparable even to Go-based applications. Notably, NestJS, when configured to use Fastify instead of Express, nearly matches the performance of standalone Fastify setups, despite its robust functionality. This demonstrates the potential for significant performance gains through strategic framework selection. AdonisJS also shows strong performance by avoiding reliance on Express.js and using a custom HTTP server.

6. Threats to validity

Conclusion validity: Concerns arise from using a custom-developed HTTP benchmarking tool over established software like JMeter, performance of which was found inadequate for our needs. Low-overhead alternatives could not capture performance changes over time, crucial for testing our hypothesis. We mitigated thhe risks by calibrating our tool, Gocannon, against established benchmarks like WRK.

Internal validity: Improvements in Node.js performance might not solely be due to V8 engine optimizations. To test this, additional SUTs in Go and Python were implemented and showed no similar performance anomalies, confirming the influence of V8's Turbofan optimizations on Node.js performance.

External validity: To generalize results beyond the experimental settings, various load testing scenarios were executed. Since all of the examined frameworks improved their performance metrics over the first seconds of the load test in every scenario, we conjecture that such performance anomaly is inherent to Node.js backend applications.

Construct validity: While aiming for realistic backend simulation, synthetic benchmarking was chosen to eliminate confounding factors, following framework best practices and utilizing recommended libraries. This approach, while potentially limiting real-world applicability, ensured controlled testing environments as outlined in Sec. 4.

7. Conclusions

In this study we examined a number of JavaScript frameworks in several testing scenarios, to evaluate their performance under load.

The contributions of the paper are two-fold. First, the results show that the performance of

Node.js backend applications improve over time under sustained load. The subject SUTs exhibited such tendency in every load testing scenario included in the experiment. The fact that such performance anomalies were not observed in baseline Python and Go SUT implementations strongly indicates that the observed peculiarity is, as previously hypothesised, caused by Node.js runtime optimization techniques used by the V8 JavaScript engine. Secondly, we also contribute a tool that solves several issues of synthetic benchmarks with measuring changes of performance in time.

Based on the observations reported in this work, we formulate the following practical recommendations for Node.js backend application developers:

- If achieving low request latency and high throughput is a strict requirement, then Fastify, Hapi, AdonisJS or NestJS should be preferred over Express.js.
- For NestJS-based applications, we advise to use a Fastify adapter instead of the default Express HTTP server implementation.
- We also advise to use least-connections load balancing algorithm instead of round-robin when deploying them behind a load balancer in an effort to prevent newly-spawned instances from becoming overloaded.
- We also recommend to avoid using Node.js for latency-critical applications for which delivering consistent performance throughout the entire process lifetime without allowing for any warm-up period under reduced load after application startup is critical.

In the future, this study could be extended by incorporating real-world applications with simulated load modelled after actual user behaviour.

References

[1] Challapalli, S. S. N., Kaushik, P., Suman, S., Shivahare, B. D., Bibhu, V., and Gupta, A. D.: Web Development and performance comparison of Web Development Technologies in Node.js and Python. In: *2021 International Conference on Technological Advancements and Innovations (ICTAI)*. 2021, pp. 303–307.

[2] Chhetri, N.: A Comparative Analysis of Node.js (Server-Side JavaScript). In: 2016. URL: https://api.semanticscholar.org/CorpusID:64435548.

[3] Draheim, D., Grundy, J., Hosking, J., Lutteroth, C., and Weber, G.: Realistic load testing of Web applications. In: *Conference on Software Maintenance and Reengineering (CSMR'06)*. 2006, 11 pp.–70.

[4] Gao, R. and Jiang, Z. M.: An Exploratory Study on Assessing the Impact of Environment Variations on the Results of Load Tests. In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 2017, pp. 379–390.

[5] Gao, R., Jiang, Z. M., and Barna Cornel an Litoiu, M.: A Framework to Evaluate the Effectiveness of Different Load Testing Analysis Techniques. In: *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 2016, pp. 22–32.

[6] Grossman, D., McCabe, M., Staton, C., Bailey, B., Frieder, O., and Roberts, D.: Performance testing a large finance application. In: *IEEE Software* 13.5 (1996), pp. 50–54.

[7] Hamed, O. and Kafri, N.: Performance testing for web based application architectures (.NET vs. Java EE). In: *2009 First International Conference on Networked Digital Technologies*. 2009, pp. 218–224.

[8] Ihrig, C.: Introduction to libuv: What's a Unicorn Velociraptor? In: *Node.js Interactive Conference*. [on-line] https://www.youtube.com/watch?v=_c51fcXRLGw. Montreal, 2019.

[9] Kasture, H. and Sanchez, D.: Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In: *2016 IEEE International Symposium on Workload Characterization (IISWC)*. 2016, pp. 1–10.

[10] Khan, R. and Amjad, M.: Smoke testing of web application based on ALM tool. In: *2016 International Conference on Computing, Communication and Automation (ICCCA)*. 2016, pp. 862–866.

[11] Khan, R. and Amjad, M.: Web application's performance testing using HP LoadRunner and CA Wily introscope tools. In: *2016 International Conference on Computing, Communication and Automation (ICCCA)*. 2016, pp. 802–806.

[12] Kolic, K., Gusev, M., and Ristov, S.: Performance analysis of a new cloud e-Business solution. In: *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 2015, pp. 263–268.

[13] Lei, K., Ma, Y., and Tan, Z.: Performance comparison and evaluation of web development technologies in php, python, and node. js. In: *2014 IEEE 17th international conference on computational science and engineering*. IEEE. 2014, pp. 661–668.

[14] Lenka, R. K., Rani Dey, M., Bhanse, P., and Barik, R. K.: Performance and Load Testing: Tools and Challenges. In: *2018 International Conference on Recent Innovations in Electrical, Electronics Communication Engineering (ICRIEECE)*. 2018, pp. 2257–2261.

[15] Meira, J. A., Almeida, E. C. d., Le Traon, Y., and Sunye, G.: Peer-to-Peer Load Testing. In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 2012, pp. 642–647.

[16] Mustafa, M. E.: Load balancing algorithms Round-Robin (RR), LeastConnection, and Least-Loaded Efficiency. In: *Computer Science & Telecommunications* 51.1 (2017).

[17] Pu, Y. and Xu, M.: Load Testing for Web Applications. In: *2009 First International Conference on Information Science and Engineering*. 2009, pp. 2954–2957.

[18] Singh, M. and Singh, R.: Load Testing of web frameworks. In: *2012 2nd IEEE International Conference on Parallel, Distributed and Grid Computing*. 2012, pp. 592–596.

[19] Stamenov, D., Venov, D., and Gusev, M.: Scalability performance evaluation of the E-Ambulance software service. In: *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 2018, pp. 0324–0329.

[20] Suffian, M. D. M. and Fahrurazi, F. R.: Performance testing: Analyzing differences of response time between performance testing tools. In: *2012 International Conference on Computer Information Science (ICCIS)*. Vol. 2. 2012, pp. 919–923.

[21] Tran, P., Gosper, J., and Gorton, I.: Evaluating the sustained performance of COTS-based messaging systems. In: *Software Testing, Verification and Reliability* 13.4 (2003), pp. 229–240.

[22] Vögele, C., Hoorn, A. van, Schulz, E., Hasselbring, W., and Krcmar, H.: WESSBAS: extraction of probabilistic workload specifications for load testing and performance prediction-a model-driven approach for session-based application systems. eng. In: *Software and systems modeling* 17.2 (2018), pp. 443–477.

[23] Zhu, L.-Z. and Li, F.: The collection and analysis of performance parameters in web applications. In: *Proceedings of 2011 International Conference on Electronics and Optoelectronics*. Vol. 3. 2011, pp. V3-150-V3–154.

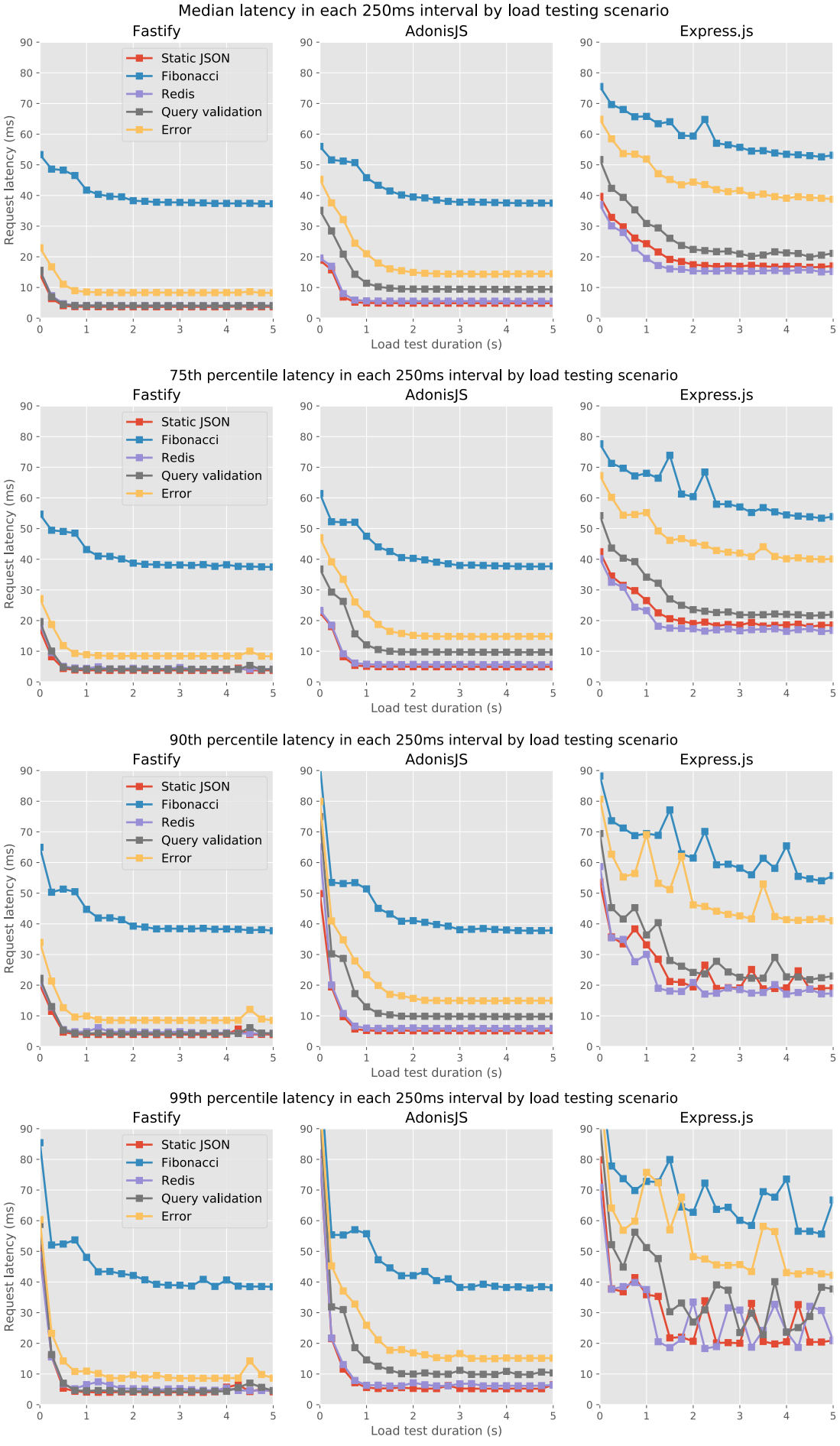

Fig. 6. Median, 75th, 90th and 99th latency percentiles of requests completed by Fastify, AdonisJS and Express.js SUTs by load testing scenario